Project Report

OTH
Amberg-Weiden

“Machine Learning (W2526) ”

Submitted by

Umang Dholakiya

Supervised by

Prof. Dr. Patrick Levi

Ostbayerische Technische Hochschule Amberg-Weiden
Department of Electrical Engineering, Media and Computer Science

January 10,2026

Abstract

This project solves a two-part tabular regression challenge with different objectives and deployment constraints. In Part 1, the goal is to learn a model that predicts the continuous variable `target01` from high-dimensional feature data and to generate reliable predictions for an unseen evaluation set in the required submission format. In Part 2, the objective changes from maximizing predictive power to producing a lightweight predictor for `target02`: the target is assumed to depend on only a few features and to follow a small number of simple conditional rules, and the final solution must run on an edge device where machine learning inference is not permitted. To meet these requirements, use a validated tree-based regression pipeline for Part 1 and, for Part 2, reverse-engineer the underlying piecewise relationship from the training data and translate it into an executable set of if–else conditions and arithmetic calculations compatible with the provided framework.

1 Introduction

The challenge is based on a tabular dataset with 10,000 samples and 273 input features, and it contains two regression problems with fundamentally different requirements. In Part 1, the goal is straightforward: learn a model that generalizes well and produces accurate predictions for `target01` on an unseen evaluation set. This part mainly rewards strong validation practice and a model choice that can handle many correlated features and non-linear relationships. In Part 2, predictive quality alone is not sufficient: `target02` must be implemented for deployment on an edge device where machine learning inference is not permitted. The task therefore becomes a structured reverse-engineering problem—identify the small subset of relevant features and express the mapping from inputs to `target02` as a small set of simple if–else conditions with numerical calculations, compatible with the provided framework and its runtime restrictions. The remainder of this report first describes the modeling pipeline used for `target01` and its validation, and then details how the rule system for `target02` was derived, implemented, and verified.

2 Part 1: Predicting target01

In Part 1,train a supervised regression model[1] for target01 using the provided training set dataset_37.csv together with its ground-truth labels from target_37.csv. The trained pipeline is then applied to the unlabeled evaluation set EVAL_37.csv to predict the target01 values (the target label) for each evaluation sample. .

2.1 Problem statement

In Part 1, the objective is to predict the continuous regression target `target01` from the provided tabular input features. The training data `dataset_37.csv` contains 10,000 samples with 273 features per sample, together with the corresponding ground-truth values of `target01` from `target_37.csv`. The high dimensionality of the feature space relative to the sample size presents a significant risk of overfitting, requiring robust feature selection. The evaluation file `EVAL_37.csv` provides the same feature representation but without targets. The required deliverable is a submission file `EVAL_target01_37.csv` containing one prediction per evaluation row, in the original order, using the header `target01`. The primary success criterion is prediction quality on the hidden evaluation labels, while the solution must also be methodologically sound (e.g., avoiding data leakage) and correctly implemented according to the required file format.

2.2 Approach (End-to-End Pipeline)

Task 1 solution follows a fixed, reproducible pipeline designed to maximize generalization on the hidden evaluation labels while avoiding common sources of leakage. The implementation is split across two scripts: `Part1.py` contains the final end-to-end training/evaluation/prediction pipeline used to generate the submission, and `parameter_experiment.py` contains a focused hyperparameter search used to identify a strong configuration for the base regressor.

1) Data loading.

Load the training feature matrix from `dataset_37.csv`, the ground-truth labels for `target01` from `target_37.csv`, and the evaluation feature matrix from `EVAL_37.csv`. Because the submission is evaluated on hidden labels, correctness of feature alignment is critical: the code checks that training and evaluation share identical feature columns in the same order. This prevents a high-impact failure mode where a model produces valid-looking outputs for the wrong feature mapping. Alternatives such as relying on implicit column order or manual column handling are error-prone and can silently corrupt predictions.

2) Data preprocessing and quality checks.

To ensure robustness against potential data quality issues, incorporated a `SimpleImputer` (median strategy) as the first step of the pipeline[2]. Although our pre-training integrity checks (e.g., for duplicates, constant columns, and missing values) confirmed that the provided training data is currently clean, this defensive step ensures the model can handle missing values gracefully during inference without crashing. This avoid aggressive scaling (e.g., `StandardScaler`) as the chosen tree-based models are invariant to monotonic feature transformations.

3) Feature selection .

The input space contains 273 features, which makes overfitting more likely because a flexible model can exploit weak or redundant signals that do not hold up on unseen data. To control this, use an embedded feature selection step implemented as `SelectFromModel` with an `ExtraTreesRegressor`. The ExtraTrees model assigns an importance score to each feature based on how much it contributes to reducing prediction error across many randomized decision trees. then after apply a median-importance threshold, keeping only the features whose importance is at least the median. In practice, this reduces the feature set from 273 to 136 features (about 50% retention). Importantly, the selector is placed inside an `sklearn Pipeline`, so during 5-fold cross-validation the feature selection[3] is re-fitted using only the training folds and then applied to the validation fold. This prevents the common leakage mistake of selecting features once on the full dataset before validation. Alternatives such as using all features without selection (higher variance and slower training) or manually removing features by intuition (hard to justify and not reproducible) were deliberately avoided.

4) Parameter optimization .

To avoid relying on arbitrary defaults, finetune the hyperparameters[4] of the HistGradientBoostingRegressor using GridSearchCV[7] in parameter_experiment.py. The search is performed over a compact, structured grid covering learning rate, number of boosting iterations, maximum leaf nodes (tree complexity), minimum samples per leaf, and L2 regularization. The search space comprised 243 parameter combinations, each evaluated with 5-fold `KFold` cross-validation. These parameters directly control the bias–variance trade-off: specifically, smaller learning rates and stronger regularization reduce overfitting, while leaf constraints limit model complexity. That's why chose a full grid search because the space is small and interpretable, ensuring transparency and reproducibility. Alternatives such as tuning on a single hold-out split (high variance) or using the evaluation set (data leakage) were rejected.

Important scope note: This tuning experiment targets the base HGBR configuration, whereas the final submitted pipeline additionally includes the embedded feature-selection step. While a joint search over the full pipeline is theoretically exhaustive, the independent tuning strategy yielded a highly stable configuration, rendering the substantial computational cost of a nested pipeline search unnecessary..

5) 5-fold cross-validation .

This estimate performance using 5-fold (k-fold) cross-validation[5] with shuffling and a fixed `random_state`. Compared to a single train/validation/test split, K-fold CV provides a lower-variance estimate of generalization and uses the full dataset more efficiently (each sample serves as validation once). This is particularly important when evaluation labels are hidden and overfitting to a single split provides misleading confidence. Alternatives such as a fixed 70/15/15 split can be acceptable when data is very large, but here it would waste data for training and produce a noisier model-selection signal.

6) Best model selection and final training.

Select the final configuration based on cross-validated performance (primary metric: RMSE, MAE and R^2) and stability across folds. After selecting the pipeline configuration, re-train it once on the full training set to maximize the data available for fitting before producing evaluation predictions. This "CV for selection, full-data fit for final model" recipe is standard for achieving the best expected performance on an unseen test distribution.

7) Evaluation and prediction generation.

For internal evaluation compute fold-wise metrics and out-of-fold predictions, which provide an unbiased estimate because each training point is predicted by a model that did not see it. Finally, apply the trained pipeline to `EVAL_37.csv` and export the prediction file `EVAL_target01_37.csv` with the required header `target01` and preserved row order.

2.3 Results (Feature Selection vs. Baseline)

To quantify the impact of feature selection, compare the final pipeline (embedded feature selection + HGBR) against an otherwise identical baseline that uses the same regressor without feature selection. Both models are evaluated under the same 5-fold `KFold` protocol, so differences can be attributed to the selection step rather than validation artifacts.

Metric	With selection	Without selection	Improvement
MAE	0.070452 ± 0.004699	0.074450 ± 0.005234	+5.37%
RMSE	0.088561 ± 0.005073	0.093234 ± 0.005229	+5.01%
R^2	0.859576 ± 0.017345	0.844368 ± 0.019170	+1.80%

Table 1: *Task 1 cross-validation results (5-fold `KFold`) with and without feature selection.*

Across all three metrics, feature selection yields a consistent improvement and does not increase fold-to-fold variance, suggesting that removing low-importance predictors reduces redundancy and improves generalization rather than introducing instability. The final submission predictions are produced by fitting the selected pipeline on the full training set and exporting one `target01` prediction per row for `EVAL_37.csv` to `EVAL_target01_37.csv` with the required header and row order.

3 Part 2: Rule Discovery and Rule-Based Deployment (target02)

3.1 Problem statement

In Part 2, the objective is to predict the continuous regression target `target02` under a strict *edge-device* constraint: the final submitted predictor must run *without any machine-learning inference at runtime*. Unlike Part 1 (where training and inference with ML models is allowed), Part 2 requires a Python file `framework_37.py` that follows the provided template and computes predictions using only (i) simple threshold conditions and (ii) arithmetic formulas.

The template evaluates a list of rule pairs `(condition, calc)`. Each `condition` is either `None` (always true) or a tuple `(feature_index, operator, threshold)`; each `calc` is a function that takes one feature row (NumPy array) and returns one numeric prediction. At runtime, the first satisfied condition wins, so ordering implements `if/elif/else` semantics.

Therefore, the core challenge is to (A) discover a compact rule system *offline* using labeled training data, and then (B) deploy the final rules inside the template using only comparisons and arithmetic (no sklearn objects, no model loading, no additional imports). The file `EVAL_37.csv` contains features only; its labels are hidden and must never be used during discovery.

3.2 Approach (Offline Rule Discovery + Rule-Based Deployment)

Task 2 follows a two-stage strategy: **learn rules offline, execute rules online**. Offline, use ML tools only to *discover* thresholds and coefficients. Online (inside `framework_37.py`), the predictor is a pure rule engine: condition checks + arithmetic.

1) Load data and ensure feature-index consistency.

The runtime framework evaluates conditions on a NumPy row vector using zero-based indexing, so the mapping must be exact: `feat_132` must correspond to `arr[132]`, etc. This is treated as a hard correctness constraint, not an implementation detail.

2) Decision tree exploration for regime structure (not the deployed predictor).

Train a decision tree regressor[8] on the labeled training data to discover which feature acts like a *switch* that separates the data into a few *value ranges* (regions) where `target02` follows different simple formulas. A tree is appropriate for this discovery phase because it explicitly searches for threshold splits and produces readable conditions of the form $\texttt{feat\_j} \leq \texttt{threshold}$[6].

3) Threshold extraction from tree nodes.

From the fitted tree, extract candidate thresholds by scanning tree nodes and collecting thresholds whenever the split feature matches the dominant gating feature. In our case, the tree consistently selects `feat_132` as the primary split feature for `target02`, indicating that `feat_132` controls which linear relationship is active.

4) Threshold simplification: from noisy tree thresholds to clean deployable boundaries.

The raw tree thresholds are not clean numbers; they include floating-point noise and multiple near-duplicate thresholds around the same boundary. For `feat_132`, the extracted unique thresholds (example values) are:

$$t = [0.2,\ 0.5004,\ 0.699,\ 0.7,\ 0.701,\ 0.7001,\ 0.7083] = [0.2,\ 0.5004,\ 0.7].$$

A naive approach is to take the first three thresholds as the region boundaries:

However, when it directly use these *raw* thresholds to define the four regions and then fit per-region linear models, the reconstruction is *near-perfect but not exact* (because some samples close to the true boundary values are assigned to the wrong region).

Raw-threshold metrics:

$R^2 = 0.999595$; MAE $= 7.393696 \times 10^{-4}$; RMSE $= 1.692485 \times 10^{-2}$; MaxAbsErr $= 1.057911$.

To produce a clean, stable, and deployable rule set, simplify thresholds by rounding to one decimal and removing duplicates. This yields the three stable boundary points:

$$0.2, \quad 0.5, \quad 0.7.$$

With these simplified thresholds, the full rule system achieves exact reconstruction up to floating-point precision (validated later):

$$R^2 = 1.0, \quad \text{with errors at } \sim 10^{-15}.$$

This is strong evidence that the true underlying rule uses clean boundaries (0.2, 0.5, 0.7), and that the extra raw thresholds are artifacts of tree optimization rather than the true generating process.

5) Region creation (four deployable regimes).

The simplified thresholds define four regions evaluated in order:

R1: $\texttt{feat\_132} \le 0.2$, R2: $0.2 < \texttt{feat\_132} \le 0.5$, R3: $0.5 < \texttt{feat\_132} \le 0.7$, R4: $\texttt{feat\_132} > 0.7$.

This mapping is chosen because it translates directly into the framework's sequential `pair_list` evaluation (equivalent to an `if/elif/elif/else` ladder).

6) Coefficient derivation via per-region `LinearRegression`[9].

Once the conditions fix the four regions, then it fit a separate linear regression model in each region to obtain a simple deployable arithmetic rule. Restrict the predictor features to:

$$\texttt{feat\_108},\ \texttt{feat\_116},\ \texttt{feat\_255},$$

because they are sufficient for exact reconstruction while keeping the deployed rule minimal (three multiplications and two additions).

Let $X \in \mathbb{R}^{n\times 3}$ be the region-specific feature matrix and $y \in \mathbb{R}^n$ be `target02` values for that region. Linear regression solves:

$$\min_{\beta\in\mathbb{R}^3} \sum_{i=1}^{n} \left(X_i\beta - y_i\right)^2,$$

which can be expressed (conceptually) by the normal equation:

$$\beta = (X^\top X)^{-1} X^\top y.$$

In practice, sklearn computes the least-squares solution using stable linear algebra routines (avoiding explicit inversion). The resulting coefficients are then treated as constants and copied into `framework_37.py` (no ML objects are used at runtime).

Region	Condition on `feat_132`	Samples	coef_108	coef_116	coef_255	R^2	Max error
1	≤ 0.2	1,999	1.35	1.75	-0.75	1.0	$\sim 10^{-15}$
2	$(0.2, 0.5]$	2,951	0.35	-0.45	0.55	1.0	$\sim 10^{-15}$
3	$(0.5, 0.7]$	2,011	0.15	0.85	-1.95	1.0	$\sim 10^{-15}$
4	> 0.7	3,039	1.85	-1.75	-0.75	1.0	$\sim 10^{-15}$

Table 2: *Region-wise summary: derived conditions, sample counts, per-region `LinearRegression` coefficients, and validation outcomes.*

7) Validation: parity checks and numerical precision.

Validate the full rule system by applying the if/elif ladder plus the region-wise linear formulas to the training set and comparing predictions to the true `target02`. Each region achieves $R^2 = 1.0$ and errors at the 10^{-15} scale, which indicates floating-point rounding rather than approximation error. This is the technical reason why perfect reconstruction is plausible and legitimate here: the target is consistent with a deterministic piecewise-linear generating function.

8) Runtime deployment in `framework_37.py` (NO ML at runtime).

The final deployment file implements:

- Conditions as tuples $(132, \leq, 0.2)$, $(132, \leq, 0.5)$, $(132, \leq, 0.7)$, and `None`,
- Four arithmetic `calc` functions implementing only the corresponding linear formulas using indices 108, 116, and 255,
- An ordered `pair_list` so the first satisfied condition is selected.

3.3 Results

The final discovered conditions and deployed rule structure are:

- **Gating feature:** `feat_132`
- **Derived thresholds:** 0.2, 0.5, 0.7
- **Derived ordered conditions:**

```
if feat_132 <= 0.2:   Region 1
elif feat_132 <= 0.5: Region 2
elif feat_132 <= 0.7: Region 3
else:                 Region 4
```

- **Predictor features per region:** `feat_108`, `feat_116`, `feat_255`

The corresponding deployed formulas are:

$$\begin{aligned}
\textbf{R1: } \texttt{feat\_132} \leq 0.2 : &\quad \texttt{target02} = 1.35 \cdot \texttt{feat\_108} + 1.75 \cdot \texttt{feat\_116} - 0.75 \cdot \texttt{feat\_255} \\
\textbf{R2: } 0.2 < \texttt{feat\_132} \leq 0.5 : &\quad \texttt{target02} = 0.35 \cdot \texttt{feat\_108} - 0.45 \cdot \texttt{feat\_116} + 0.55 \cdot \texttt{feat\_255} \\
\textbf{R3: } 0.5 < \texttt{feat\_132} \leq 0.7 : &\quad \texttt{target02} = 0.15 \cdot \texttt{feat\_108} + 0.85 \cdot \texttt{feat\_116} - 1.95 \cdot \texttt{feat\_255} \\
\textbf{R4: } \texttt{feat\_132} > 0.7 : &\quad \texttt{target02} = 1.85 \cdot \texttt{feat\_108} - 1.75 \cdot \texttt{feat\_116} - 0.75 \cdot \texttt{feat\_255}.
\end{aligned}$$

On the labeled training set, the full rule system reconstructs `target02` up to floating-point precision:

- Max absolute error $\sim 10^{-15}$ (floating-point rounding scale)
- Region-wise $R^2 = 1.0$ for all four regions (Table 2)

Worked prediction example (step-by-step). Consider one sample with:

$$\texttt{feat\_108} = 0.9886, \quad \texttt{feat\_116} = 0.7548, \quad \texttt{feat\_132} = 0.1054, \quad \texttt{feat\_255} = 0.8282.$$

Step 1 (region selection): since $\texttt{feat\_132} = 0.1054 \leq 0.2$, the sample falls in Region 1.
Step 2 (apply Region 1 formula):

$$\hat{y} = 1.35 \cdot \texttt{feat\_108} + 1.75 \cdot \texttt{feat\_116} - 0.75 \cdot \texttt{feat\_255}.$$

Step 3 (compute contributions):

$$1.35(0.9886) = 1.3346, \quad 1.75(0.7548) = 1.3209, \quad -0.75(0.8282) = -0.6211.$$

Step 4 (sum):

$$\hat{y} = 1.3346 + 1.3209 - 0.6211 = 2.0344.$$

This matches the stored `target02` for the same row up to floating-point rounding (confirmed by the full-dataset parity check).

4 Conclusion

This project solved both parts of the Machine Learning Challenge for ID 37. For Part 1, we built a reproducible regression pipeline for target01 using 5-fold cross-validation to guide model selection and avoid leakage, then trained the final model on the full dataset and produced EVAL_target01_37.csv in the required format. For Part 2, we reverse-engineered target02 into a compact, interpretable rule system by identifying threshold boundaries on feat_132 and fitting per-region linear coefficients, then deployed the final logic in framework_37.py using only ordered conditions and arithmetic (no ML at runtime). Overall, the submissions are accurate, compliant with the provided interfaces, and designed to be both reliable for hidden evaluation and lightweight for edge deployment.

References

[1] N. A. Kumar, L. Jangale, V. Sathe, A. Shelke, and T. Redij, "Study of Supervised Logistic Regression Algorithm," *Alochana Journal*, vol. 13, no. 11, pp. 227–230, 2024, doi: 20.14118.AJ.2024.V13I11.2806. [Online]. Available: https://www.researchgate.net/publication/385985292_Study_of_Supervised_Logistic_Regression_Algorithm. Accessed: Jan. 12, 2026.

[2] K. Maharana, S. Mondal, and B. Nemade, "A review: Data pre-processing and data augmentation techniques," *Global Transitions Proceedings*, vol. 3, no. 1, pp. 91–99, 2022, doi: 10.1016/j.gltp.2022.04.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666285X22000565. Accessed: Jan. 12, 2026.

[3] J. Cai, J. Luo, S. Wang, and S. Yang, "Feature selection in machine learning: A new perspective," *Neurocomputing*, vol. 300, pp. 70–79, 2018, doi: 10.1016/j.neucom.2017.11.077. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218302911. Accessed: Jan. 12, 2026.

[4] L. Franceschi, M. Donini, V. Perrone, A. Klein, C. Archambeau, M. Seeger, M. Pontil, and P. Frasconi, "Hyperparameter Optimization in Machine Learning," *Foundations and Trends® in Machine Learning*, vol. 18, no. 6, pp. 1054–1201, 2025, doi: 10.1561/2200000088. [Online]. Available: https://arxiv.org/abs/2410.22854. Accessed: Jan. 12, 2026.

[5] N. Darapureddy, N. Karatapu, and T. K. Battula, "Research of Machine Learning Algorithms using K-Fold Cross Validation," *International Journal of Engineering and Advanced Technology (IJEAT)*, vol. 8, no. 6S, pp. 216–218, Aug. 2019, doi: 10.35940/ijeat.F1043.0886S19. [Online]. Available: https://www.ijeat.org/wp-content/uploads/papers/v8i6S/F10430886S19.pdf. Accessed: Jan. 12, 2026.

[6] B. Anderson, C. Storlie, M. Yates, and A. McPhall, "Automating reverse engineering with machine learning techniques," in *Proceedings of the 2014 ACM Workshop on Artificial Intelligence and Security (AISec '14)*, pp. 103–112, Nov. 2014. Association for Computing Machinery, doi: 10.1145/2666652.2666665. [Online]. Available: https://mayoclinic.elsevierpure.com/en/publications/automating-reverse-engineering-with-machine-learning-techniques/. Accessed: Jan. 12, 2026.

[7] "Optimization of Regression Models Using Machine Learning: A Comprehensive Study with Scikit-learn," ResearchGate publication. [Online]. Available: https://www.researchgate.net/publication/384051854_Optimization_of_Regression_Models_Using_Machine_Learning_A_Comprehensive_Study_with_Scikit-learn. Accessed: Jan. 12, 2026.

[8] "A Survey of Decision Trees: Concepts, Algorithms and Applications," ResearchGate publication. [Online]. Available: https://www.researchgate.net/publication/381564302_A_Survey_of_Decision_Trees_Concepts_Algorithms_and_Applications. Accessed: Jan. 12, 2026.

[9] "A Review on Linear Regression: Comprehensive in Machine Learning," ResearchGate publication. [Online]. Available: https://www.researchgate.net/publication/348111996_A_Review_on_Linear_Regression_Comprehensive_in_Machine_Learning. Accessed: Jan. 12, 2026.